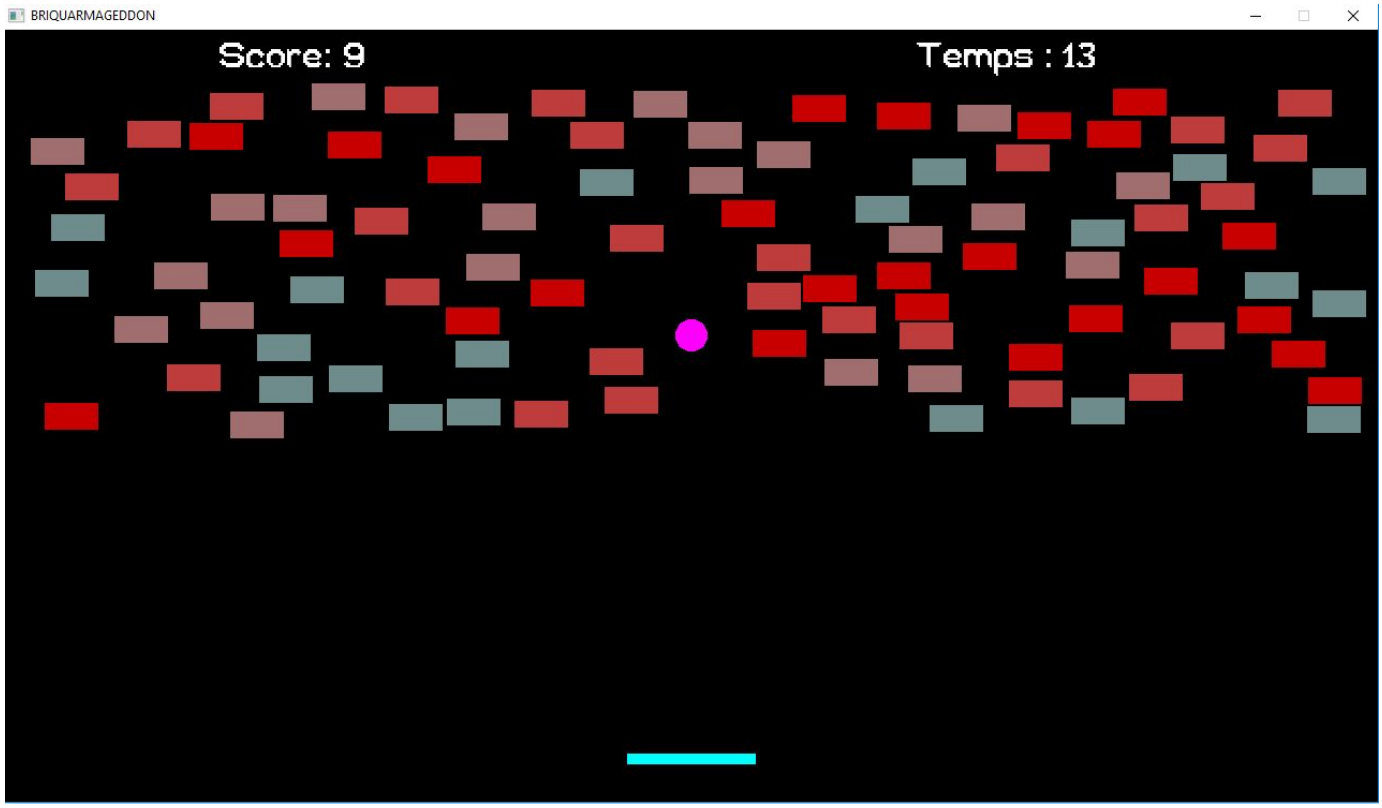
Projet Programmation :
Casse Brique en langage C++



Geipi 2A IT

Anquetin Romain
Waldner Céline

Sommaire

Introduction

Dans le cadre des projets de programmation des GEIPI IT, nous avons réalisé un jeu style casse brique en c++. Pour les graphismes du jeu nous avons utilisé la bibliothèque graphique SFML. Nous avons souhaité faire une version assez basique mais fonctionnelle du jeu dans l'optique de pouvoir rajouter plus tard d'autres options plus poussées, par exemple rajouter des éléments de gameplay comme des pouvoir utilisables.
Le but de ce jeu est de casser toutes les briques affichées à l'écran à l'aide d'une balle que l'on fera rebondir sur une raquette. Les briques seront positionnés aléatoirement sur l'espace. Le joueur pourra déplacer la raquette de gauche à droite de l'écran afin de faire rebondir la balle, si la balle ne rebondit pas sur la raquette et qu'elle tombe, elle est perdue. A chaque brique touchée, le score qui sera également affiché sur l'écran, augmentera. Si toutes les briques ont étés détruites ou que toutes les vies ont étés épuisées, la partie se termine. Nous allons voir toutes ces fonctionnalités plus en détails, en expliquant les différentes classes et leur rôle.

Diagramme du jeu global

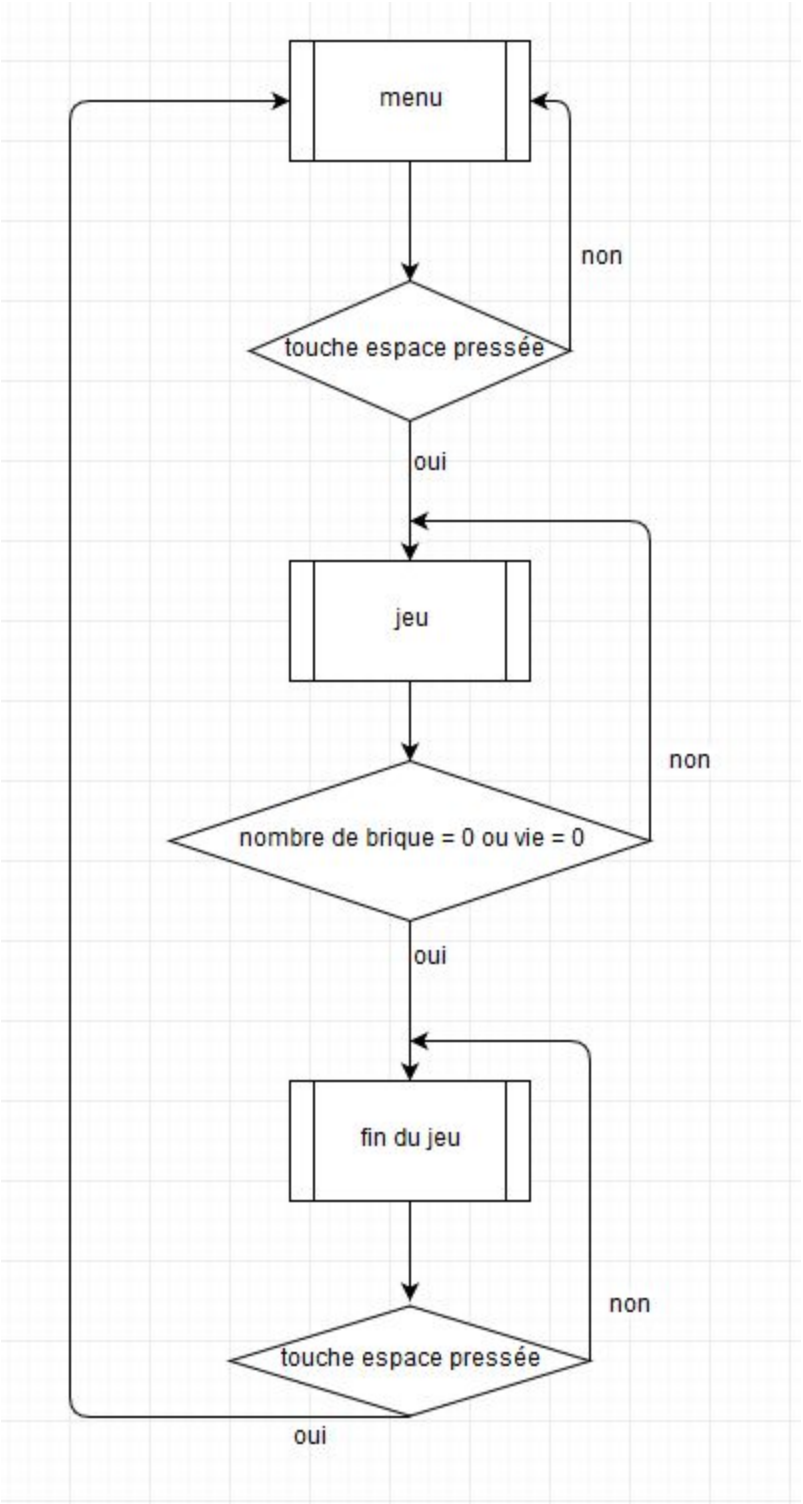
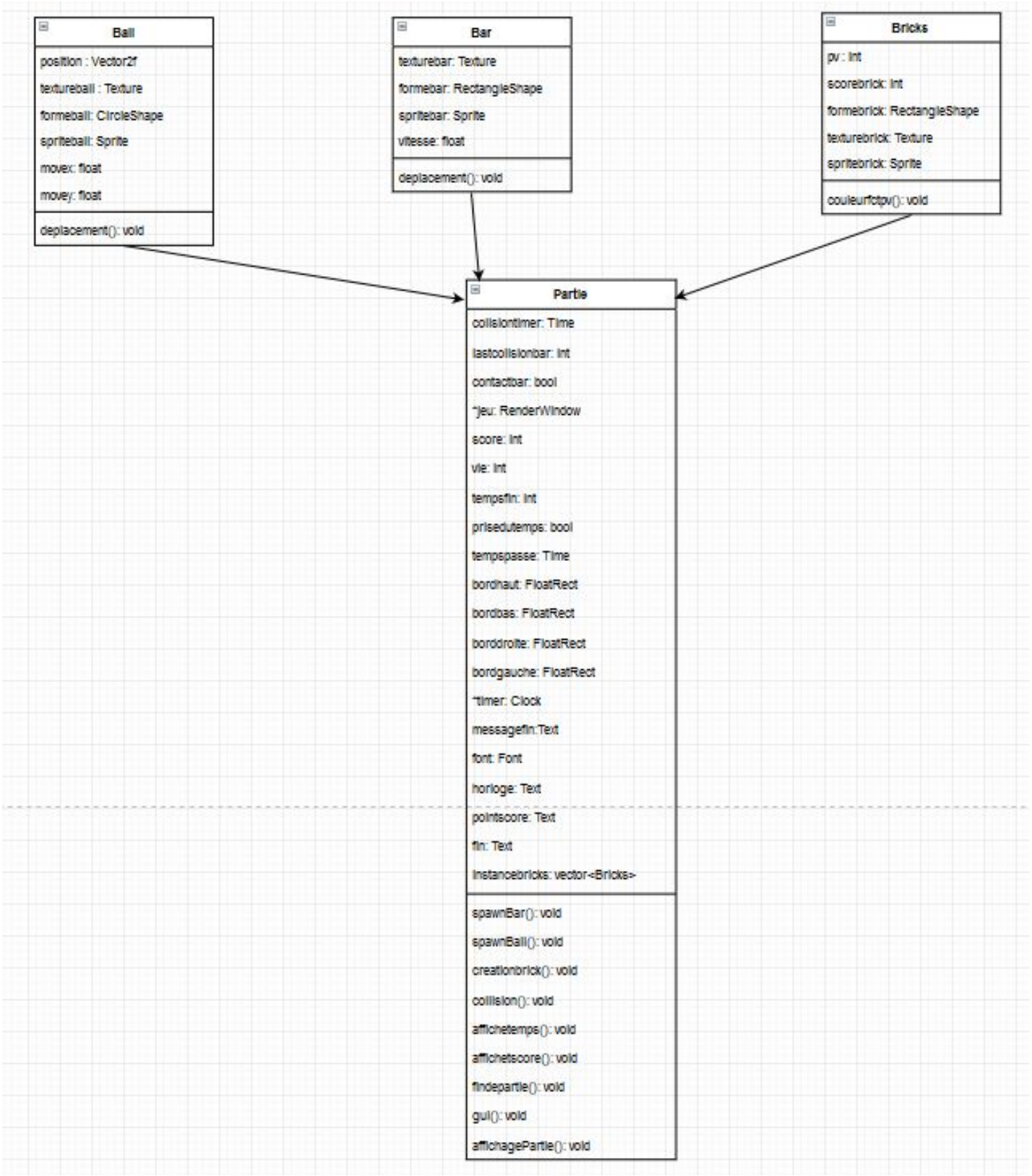


Diagramme des classes



1) Explication des classes Bar, Ball et Bricks

Nous avons construit notre code avec quatre classes, nous allons voir maintenant trois d'entre elles.

a) La classe Brick

Aperçu de Bricks.h

```
#ifndef BRICKS_H
#define BRICKS_H
#include <SFML/Graphics.hpp>
#include <iostream>
class Bricks
{
public:
    int pv, scorebrick;
    sf::RectangleShape formebrick;
    sf::Texture texturebrick;
    sf::Sprite spritebrick;
    sf::Color couleurbrick;

    void couleurfctpv();

    Bricks();
```

La classe Bricks permet de créer des objets Bricks pour l'utiliser dans la classe Partie, nous avons besoin que les briques aient des points de vie pour pouvoir les détruire, une couleur pour indiquer leur nombre de point de vie restant et une forme rectangulaire pour pouvoir l'afficher à l'écran et travailler avec des collisions. Nous avons aussi prévu que la brique puisse avoir une texture mais nous avons choisis de ne pas nous en servir pour que le jeu reste plus lisible et léger.

La méthode **couleurfctpv()** permet d'attribuer une couleur à chaque brique en fonction de ses points de vie du bleu pour les briques qui ont peu de points de vie vers le rouge en passant par le vert.

b) La classe Bar

Aperçu de Bar.h

```
class Bar
{
    public:
        sf::Texture texturebar;
        sf::RectangleShape formebar;
        sf::Vector2f spawnpointball;
        sf::Sprite spritebar;
        float vitesse;

void deplacement(bool droite , bool gauche);

Bar();
```

La classe Bar nous permet de créer une barre qui sera utilisée par le joueur pour éviter que la balle tombe hors des limites du jeu, l'objet barre possède une couleur, une vitesse, une forme rectangulaire pour pouvoir l'afficher à l'écran et travailler avec des collisions.

La méthode déplacement permet de déplacer la barre de gauche ou à droite avec une vitesse, elle prend en paramètre 2 booléens qui seront modifiés quand le joueur appuis sur les touches de déplacement.

Voici la méthode de déplacement pour la barre.

```
void Bar::deplacement(bool droite , bool gauche){
    if (droite){
      formebar.move(vitesse,0);

    }

    if (gauche){
       formebar.move(-vitesse,0);
    }
```

c) La classe Ball

Aperçu de Ball.h

```
#ifndef BALL_H
#define BALL_H
#include <SFML/Graphics.hpp>


class Ball
{
public:
// attributs

    int ptvie;
    float movex, movey;


    sf::Vector2f position;
    sf::Sprite spriteball;
    sf::Texture textureball;
    sf::CircleShape formeball;
    sf::Color colball;
    void deplacement();


    Ball();
```

Cette classe nous permet de construire une balle pour le jeu. Cette balle possède les attributs **movex** et **movey** cela va nous permettre de définir la vitesse sur l'axe x et l'axe y et nous avons un attribut **ptvie** que nous n'utilisons pas mais qui nous permettra de rajouter des mécaniques de jeu plus tard.

La méthode **move()** permet de faire bouger la balle selon l'axe x et l'axe y.

Voici la fonction de déplacement pour la balle.

```
void Ball::deplacement(){
    formeball.move(movex, movey);

}
```

Ces trois classes permettent de créer des objets que la classe partie pourra utiliser. La classe Partie est notre classe principale, c'est elle qui gère tous les éléments du jeu. Nous allons maintenant voir comment.

2)Explication de la classe Partie

Aperçu de Partie. h

```
#ifndef PARTIE_H
#define PARTIE_H
#include "Bar.h"
#include "Ball.h"
#include "Bricks.h"
#include <iostream>


class Partie
{
public:

    int lastcolisionbar;
    int score;
    int vie;
    int tempsfin;
    sf::RenderWindow *jeu;
    bool prisedutemps;
    sf::Time tempspasse;
    sf::Time colisiontimer ;
    sf::FloatRect bordhaut;
    sf::FloatRect bordbas;
    sf::FloatRect borddroite;
    sf::FloatRect bordgauche;
    sf::Clock *timer;
    sf::Font font;
    Bar bar;
    Ball ball;
    std::vector<Bricks> instancebricks;

    void spawnBar();
    void spawnBall();
    void creationbrick();
    void collision();
    void affichetemps();
    void affichescore();
    void findepartie();
    void gui();
    void affichagePartie();
    void debutdepartie();

    Partie(sf::RenderWindow &gameWindow, sf::Clock &gametimer,int nbbrique);
};

#endif // PARTIE_H
```

Dans la classe Partie nous avons plusieurs attributs nous avons toutes les variables qui appartiennent au jeu comme la **vie** du joueur et le **score,** nous utilisons **lastcolisionbar** pour savoir la dernière fois que la balle a touché la barre et **tempsfin** conserve le temps que le joueur a mis pour finir la partie.
Les attributs **tempspasse** et **colisiontimer** permettent de récupérer un temps qui est définit par le pointeur ***timer** qui pointe sur une horloge qui est créée dans le **main()**.
Le booléen **prisedutemps** nous permet de récupèrer le temps une seule fois quand la partie est finie.
Les **bordhaut**, **bordbas**, **borddroite** et **bordgauche** représente les limites de la zone de jeu ils servent à éviter que la balle sorte de l'écran. L'attribut **font** représente la police de caractère utiliser pour écrire les textes.
Les objets des classes **Bar** et **Ball** sont respectivement **bar** et **ball**. L'attribut **instancebricks** est un tableau d'instance de la classe Bricks.

Les méthodes **spawnBar()** et **spawnBall()** permettent de calculer et placer où les objets doivent apparaître pour la première fois par exemple **spawnBall()** permet de calculer le point au-dessus de la barre et de déplacer la balle a ce point.
Les méthode **affichetemps()** et **affichescore()** permet d'afficher le temps sur l'écran et le score, la fonction **gui()** a le même rôle.

a) Méthode **creationbrick()**

La méthode **creationbrick()** permet de placer les briques au bon endroit et de leur donner une couleur, les briques sont générées dans une zone définie , de manière aléatoire et les briques qui ne sont pas conforme sont supprimées, nous avons choisi cette solution pour que chaque partie soit unique du point de vue du placement des briques.

```
void Partie::creationbrick()
{
    int i, j;
    srand(time(NULL));
    std::default_random_engine gen (rand());
    int positionbrickx;
    int positionbricky;
    int taillefenetrex = jeu->getSize().x - 60;
    int taillefenetrey = jeu->getSize().y/2;
    std::uniform_int_distribution<int> randx(10, taillefenetrex);
    std::uniform_int_distribution<int> randy(50, taillefenetrey);


    for (i=0; i<instancebricks.size(); i++)
    {
        positionbrickx = randx(gen);
        positionbricky = randy(gen);
        instancebricks[i].formebrick.setPosition(positionbrickx, positionbricky);
        instancebricks[i].pv=(rand()%4)+1;
        instancebricks[i].couleurfctpv();
        instancebricks[i].formebrick.setOutlineColor(sf::Color(255,255,255,0));
        instancebricks[i].formebrick.setOutlineThickness(1);
    }
    for(i=instancebricks.size()-1; i>=0; i--)
    {
        for(j=0; j<i; j++)
        {
            if(instancebricks[j].formebrick.getGlobalBounds().intersects(instancebricks[i].formebrick.getGlobalBounds()))
            {
                instancebricks.erase(instancebricks.begin()+j);
                j=i;


            }

        }
    }

}
```

Au début de la fonction nous initialisons deux générateurs de nombres aléatoires un pour l'axe x et l'autre pour l'axe y et la taille de la fenêtre ou les brique peuvent apparaître.
La première boucle permet de générer les nombres aléatoires dans la zone d'apparition et les place, leur donne des point de vie (**pv**), leur donne une couleur avec **couleurfctpv()**.
On définit un contour pour les briques d'épaisseur de 1 pixel et de couleur blanc avec un alpha à 0 donc transparent nous avons fait cela pour que le joueur n'a pas l'impression que les briques se touchent.
Ensuite dans la deuxième boucle on regarde si des briques se superposent entre elles et une des deux briques qui se superposent est supprimées.

b) Méthode **collision()**

b.1) Collision avec les bords de l'écran

Nous avons créé dans un premier temps des rectangles tout autour de la fenêtre afin de délimiter la zone de jeu, leur création se fait grâce au constructeur de la classe. On regarde la boîte de collision des objets grâce à la fonction **getGlobalBounds()** et on vérifie si elles se touchent grâce à la fonction **intersects()**. Pour les bords latéraux s'il y a contact on va tout simplement ajouter un signe moins sur la valeur du déplacement selon x. Pour le bord haut c'est la même chose mais pour la composante y. Enfin, pour le bord du bas la balle réapparaît au-dessus de la barre et on perd une des trois vies que l'on possède au départ.

Exemple de test de collision

```
if(ball.formeball.getGlobalBounds().intersects(borddroite)){
    ball.movex = - ball.movex;

}
```

b.1) Collision avec les briques et la balle

Pour les collisions avec les briques nous utilisons la fonction **intersects()** avec 2 paramètres le premier qui est l'objet à tester et le deuxième qui est un rectangle qui représente la surface qui se superpose entre les 2 objets a partir de là on regarde si le rectangle est plus grand en hauteur qu'en largeur alors la balle est entrée en contacte avec une surface verticale et donc on inverse la composante y de la vitesse de la balle sinon la balle a touchée une surface horizontale est on a à inversé la composante x de la vitesse.
Puisque qu'il y a eu une collision avec une brique on doit ajouter 1 au score et retirer 1 point de vie à la brique et recalculer sa couleur.

```
for(i=0; i<instancebricks.size(); i++)
{
    if(ball.formeball.getGlobalBounds().intersects(instancebricks[i].formebrick.getGlobalBounds(),intersection))
    {
        if(intersection.height>intersection.width)
        {
            ball.movex= -1*ball.movex;
            ball.movey= 1*ball.movey;
        }
        else
        {
            ball.movex= 1*ball.movex;
            ball.movey= -1*ball.movey;
        }

        score++;

        instancebricks[i].pv--;
        if(instancebricks[i].pv <= 0)
        {
            instancebricks.erase(instancebricks.begin()+i);
        }
        else
        {
            instancebricks[i].couleurfctpv();
        }

    }

}
```

c) Les Méthodes d'affichages

Elles concernent plusieurs méthodes permettant d'afficher des information pour le joueur à l'écran.
Pour afficher du texte à l'écran nous utilisons plusieurs fonctions. Tout d'abord pour le texte nous avons besoin de charger une police d'écriture avec la fonction **loadFromFile**. Nous devons définir la taille de l'écriture grâce à la fonction **setCharacterSize**. Nous utilisons **setFillColor** pour définir la couleur de l'écriture. La fonction **setPosition** permet de choisir l'emplacement du texte, **setString** permet d'écrire ce que le message qu'on veut afficher et **setStyle** permet de modifier le style d'écriture en gras par exemple. Enfin, **draw** permet d'afficher sur la fenêtre voulue.

c.1) Méthode **debutdepartie()**

Cette méthode est le menu de début du jeu, elle permet d'afficher le titre du jeu et un message invitant à lancer le jeu en appuyant sur la touche espace, ce mécanisme sera expliqué dans le paragraphe sur le **main()**.

Aperçu du menu du jeu :

c.2) Méthode **gui()**

Cette méthode permet d'afficher pendant toute la durée du jeu le score augmentant quand une brique est touchée et un chronomètre affichant les seconde écoulé depuis le lancement de la partie. On récupère le temps écoulé depuis le lancement du jeu grâce à la fonction **getElapsedTime**.

```
190   void Partie::gui()
191   {
192       sf::Text horloge;
193       sf::Text pointscore;
194       horloge.setFont(font);
195       pointscore.setFont(font);
196       horloge.setCharacterSize(30);
197       pointscore.setCharacterSize(30);
198       horloge.setFillColor(sf::Color::White);
199       pointscore.setFillColor(sf::Color::White);
200       tempspasse = timer->getElapsedTime();
201       horloge.setString("Temps : "+std::to_string(static_cast<int>(tempspasse.asSeconds())));
202       pointscore.setString("Score: "+std::to_string(static_cast<int>(score)));
203       horloge.setPosition(850,5);
204       pointscore.setPosition(200,5);
205       jeu->draw(horloge);
206       jeu->draw(pointscore);
207
208   }
```

Les ligne 192 à 199 permettent d'initialiser **horloge** et **pointscore** ensuite on récupère le temps qui c'est écoulé ligne 200 et on définit les textes de **horloge** avec **setString()** est en paramètre le temps écoulé on fait de même avec score mais on prend la variable **score** au lieu du temps.
draw() permet d'afficher le texte sur l'écran.

c.3) Méthode **findepartie()**

Lorsque la partie se termine on affiche le score et le temps qu'a duré la partie. On affiche également le message "GAGNE" si toutes les briques ont été détruites ou "PERDU" si la vie arrive à zéro. Un message invitant à revenir au menu pour relancer une partie s'affiche également.

Aperçu de la fin d'une partie

d) Constructeur de Partie

```
308   Partie::Partie(sf::RenderWindow &gameWindow, sf::Clock &gametimer,int nbbrique)
309   {
310       font.loadFromFile("Xeliard.ttf");
311       prisedutemps=true;
312       timer = &gametimer;
313       jeu = &gameWindow;
314       score = 0;
315       vie = 3;
316       lastcolisionbar=colisiontimer.asMilliseconds();
317       instancebricks = std::vector<Bricks>(nbbrique);
318       for(int i=0; i<instancebricks.size(); i++)
319       {
320           instancebricks[i].formebrick.setSize(sf::Vector2f(50, 25));
321       }
322       creationbrick();
323       bar.formebar.setSize(sf::Vector2f(120, 10));
324       bar.formebar.setOrigin(60, 5);
325       bar.formebar.setFillColor(sf::Color::Cyan);
326       spawnBar();
327       ball.formeball.setRadius(15);
328       ball.formeball.setOrigin(15, 15);
329       ball.formeball.setFillColor(sf::Color::Magenta);
330       spawnBall();
331       ball.movex = 0;
332       ball.movey = 5;
333       bar.vitesse = 8;
334       bordbas = sf::FloatRect(sf::Vector2f(0, jeu->getSize().y), sf::Vector2f(jeu->getSize().x, 20));
335       borddroite = sf::FloatRect(sf::Vector2f(jeu->getSize().x, 0), sf::Vector2f(20, jeu->getSize().y));
336       bordgauche = sf::FloatRect(sf::Vector2f(0, 0), sf::Vector2f( -10, jeu->getSize().y));
337       bordhaut = sf::FloatRect(sf::Vector2f(0, -20), sf::Vector2f(jeu->getSize().x, 20));
338
339   }
```

Voici le constructeur de la classe **Partie** , c'est ici que sont créés et définit tous les objets.

3) Main

La fonction main est construite sur le principe d'une machine d'état en 3 état ,le menu , le jeu et la fin de partie

a)1er état

Le 1er état est le menu de début on utilise la fonction **debutdepartie()** de la classe Partie ici instancier sous le nom d'objet **game**.

Ensuite on regarde si la barre d'espace est activée au front montant donc quand la barre d'espace vient d'être appuyée

```
if (menudebut==true)
{
    std::cout <<"menu"<<std::endl;
    game.debutdepartie();
    if (sf::Keyboard::isKeyPressed(sf::Keyboard::Space)==true && triggerfrontmontant==false)
    {
        menudebut=false;
        lancementdepartie=true;
    }
    triggerfrontmontant=sf::Keyboard::isKeyPressed(sf::Keyboard::Space);
}
```

b)2eme état

```
if (lancementdepartie==true)
{
    game=Partie(renderWindow,timer,300);
    lancementdepartie=false;
    partieencours=true;
    timer.restart();
    std::cout <<"initialisation fini"<<std::endl;
}
else
{
```

Ici on efface la partie d'avant est on la recrée grâce au constructeur, on en profite aussi pour redémarrer l'horloge pour compter le nombre de seconde écoulées depuis le début de la partie

```
}
else
{
    if (partieencours==true)
    {
        game.ball.deplacement();

        if (sf::Keyboard::isKeyPressed(sf::Keyboard::D))
        {
            droite=true;
        }
        else
        {
            droite=false;
        }
        if (sf::Keyboard::isKeyPressed(sf::Keyboard::Q))
        {
            gauche=true;
        }
        else
        {
            gauche=false;
        }
        game.bar.deplacement(droite,gauche);
        game.collision();
        game.affichagePartie();
        game.gui();
        if(game.vie<=0 || game.instancebricks.size()<=0 )
        {
            findepartie=true;
            partieencours=false;
        }
    }
```

Voici le corp du jeu on déplace la balle au tout début ensuite on regarde les entrées que le joueur envoie au jeu là on va regarder les touche Q et D du clavier et on va mettre à jour des booléens pour que la fonction **deplacement()** ai les bons paramètres , ensuite nous avons la fonction **collision() , affichagePartie()** nous permet d'afficher toutes les briques ,la balle et la barre en une ligne pour plus de lisibilité , ensuite nous affichons le score et le temps avec la fonction **gui()**.
Et pour finir on regarde les conditions de victoire ou de défaite.

4) Problèmes rencontrés

Développer un jeu prend énormément de temps et le casse brique, d'apparence simple à programmer, nous auras posé quelques difficultées. Nous avons rencontré plusieurs problèmes le premier étant la nouvelle bibliothèque à apprendre, qui est la SFML. Cette bibliothèque nous a apporté beaucoup de fonctionnalités, cependant il nous a fallu du temps et beaucoup de recherches pour la maîtriser.
Nous avons aussi rencontré un problème au niveau des collisions sur comment nous allons les faire fonctionner de manière consistante.

5) Conclusion

Notre objectif de départ était de faire un jeu casse brique , nous avons réussi à faire ce jeu ,même si le rendu final n'est pas impressionnant et que quelques fonctionnalités ne sont pas disponible comme de pouvoir mettre le jeu en pause. Nous avons laissé la possibilité de continuer à développer le jeu et à ajouter du contenu. Nous avons rencontré un certains nombre de difficultés comme citées juste avant mais le jeu est fonctionnel, ce qui était notre principal but. Nous avons beaucoup appris en développant ce jeu, notamment sur la bibliothèque SFML que nous avons dû apprendre à utiliser.